# Experiments, Distributions, and Simulation

2022

## Real World Example - Design of Experiments

I currently work at a B2B technology company specializing in conversational analytics. From a a business perspective, experiment design is important when building generalized models (or studies) that span across industries. If we just took the data "as-is", the results of the model would not be valid, and could even be misleading! Some of the reasons why a poorly designed study would yield biased results are listed below:

1. Some companies have volume that is orders of magnitude higher than others (Fortune 500 vs startup)
2. Different industries have vastly different service structure (Bank vs Hardware Store)
3. Sales interactions are very different from Customer Service interactions
4. Some businesses are seasonal, some are cyclical, some are uniform
5. Etc.

In order for our model to be valid, we need to ensure we control for as many biased and confounding factors as we can, while still keeping meaningful features. To do this, techniques like Stratification, Blocking, Randomization, and Weighting are applied.

## Factorial Design Implementation

In this section, we explore the concept of factorial design with the goal of reducing a set of 10 binary features into 16 options to be shown in a survey. We first note that, in this case, Full Factorial design is impractical given that there are 1024 possible feature combinations ($2^{10}$).

```
library(FrF2)

homeFeat = FrF2(16, 10, default.levels = c('Yes', 'No'))
homeFeat
```

```
##      A   B   C   D   E   F   G   H   J   K
## 1  Yes Yes Yes Yes  No  No  No  No Yes  No
## 2   No  No  No  No  No  No  No  No  No  No
## 3   No Yes  No Yes Yes  No Yes Yes  No  No
## 4   No  No  No Yes  No  No  No Yes Yes Yes
## 5  Yes  No Yes  No Yes  No Yes Yes Yes  No
## 6  Yes Yes  No Yes  No Yes Yes  No  No Yes
## 7  Yes  No Yes Yes Yes  No Yes  No  No Yes
## 8   No  No Yes  No  No Yes Yes  No Yes Yes
## 9  Yes  No  No  No Yes Yes  No Yes  No Yes
## 10 Yes Yes  No  No  No Yes Yes Yes Yes  No
## 11  No  No Yes Yes  No Yes Yes Yes  No  No
## 12 Yes  No  No Yes Yes Yes  No  No Yes  No
## 13  No Yes Yes  No Yes Yes  No  No  No  No
## 14  No Yes Yes Yes Yes Yes  No Yes Yes Yes
## 15  No Yes  No  No Yes  No Yes  No Yes Yes
## 16 Yes Yes Yes  No  No  No  No Yes  No Yes
## class=design, type= FrF2
```

## Question 13.1

In this section we give a pair of examples of data that is expected to follow some common distributions.

1. Binomial
   a. Throwing a pair of dice - e.g P(sum > 6)
   b. Drawing cards from a deck - e.g P(drawing a Spade)
2. Geometric
   a. Baby gender - e.g P(boy after having 2 girls)
   b. Shooting a basketball - e.g P(making 1st try)
   - NOTE: The same data can be framed as a Binomial or a Geometric problem! It depends on the question.
3. Poisson
   a. Website visitors per hour - e.g P(105 visitors if $\lambda$=100)
   b. Number of credit card applications per day - e.g P(1000 applications if $\Lambda$=500)
4. Exponential
   a. Ticket purchases - e.g P(50 days prior)
   b. Demand for a new videogame e.g P(50k sold)
5. Weibull
   a. Lifetime of electronics - e.g P(screen lasting 10000 hours)
   b. Car Maintenance - e.g P(breakdown within 6 months)
   - NOTE: Again, the same data can be framed in different ways! The Weibull distribution can, in fact, transform into a Normal or an Exponential distributions with the right parameters!

## Simulation of an Airport

In this section we simulate a 2-step airport queue. The goal of the simulation is to find the number of Service representatives and the number of Self-serve scanners to keep the long term wait time under 15 minutes.

Here is as snippet of how the simulation system works:

- P1 arrives at t = [0.16500462]
- P2 arrives at t = [0.17214336]
- P3 arrives at t = [0.37305445]
- P1 time in ServiceQueue = [0.8528443]
- P1 time in ScanQueue = [0.87501753]
- P1 leaves at [1.04002215]
- P4 arrives at t = [1.10415736]
- P5 arrives at t = [1.66829429]
- P2 time in ScanQueue = [1.21356498]
- P2 leaves at [2.25358713]
- P6 arrives at t = [2.3361185]
- P3 time in ScanQueue = [0.50517184]
- P3 leaves at [2.75875897]

The code below shows how we can accomplish such a simulation in Python. Key results for t=120 (2 hours) are shown below:

1. 1 rep, 1 scanner = 45.83 minute wait
2. 2 rep, 2 scanner = 35.67 minute wait
3. 5 rep, 5 scanner = 3.5 minute wait
4. 4 rep, 4 scanner = 14.4 minute wait

For this set of runs, it looks like 4 reps and 4 scanners give us the best balance of resources vs wait time. Obviously, these results are not fully optimized, but it gives us an idea of what to expect.

```python
from random import uniform
import simpy
import numpy as np
import pandas as pd
from scipy.stats import uniform
from scipy.stats import expon


# Parameters
serviceRep_count = 4
pCheck_count = 4
pArrival_time  = .2
repService_time = .75

Sim_Time  = 100
in_system = []

def p_arrival(env, serviceRep_count, pCheck_count):
    # Assign counts to passengers
    p_counter = 0
    while True:
       # Exponential generator for arrivals
       next_p = expon.rvs(scale = pArrival_time, size = 1)
       # Wait for next person to arrive
       yield env.timeout(next_p)

       arrival_time = env.now
       p_counter += 1

       env.process(pServiceQueue(env, serviceRep_count, p_counter))
       env.process(pScanningQueue(env, pCheck_count, p_counter, arrival_time))

def pServiceQueue(env, serviceRep_count, p_number):
    with service_line.request() as req:
        yield req
        # Exponential distribution for the service process
        serviced = expon.rvs(scale = repService_time, size = 1)
        yield env.timeout(serviced)

        time_service = serviced


def pScanningQueue(env, pCheck_count, p_number, arrival_time):
    with scan_line.request() as req:
        yield req
        # Uniform distribution for the service process
        scanned = uniform.rvs(.5, 1, size = 1)
        yield env.timeout(scanned)

        time_scan = scanned
        departure_time = env.now
        system_time = departure_time[0] - arrival_time[0]
        in_system.append(system_time)
```

```
# Start environment
env = simpy.Environment()

# define resources
service_line = simpy.Resource(env, capacity = serviceRep_count)
scan_line = simpy.Resource(env, capacity = pCheck_count)

## defining the FULL arrival process
env.process(p_arrival(env, service_line, scan_line))

## run the simultion
```

```
## <Process(p_arrival) object at 0x18ec3227e50>
```

```
env.run(until = Sim_Time)

#Wait time
avg_delay = np.mean(in_system)
print('The average delay in system is {}'.format(avg_delay))
```

```
## The average delay in system is 10.37708197005925
```